System Call - [Link](https://so.v2.cs.unibo.it/wiki/index.php/Il_%27%27catalogo%27%27_delle_System_Call)

Le più importanti di UNIX sin dal principio, ossia quelle per gestire i processi; la principale funzione di un sistema di un sistema operativo è l’astrazione per creare i processi;

\*\*\* gestione processi/esecuzione

**Come viene identificato un processo?**

In sistemi Unix, i processi vengono creati principalmente utilizzando syscall fork() o [clone()](https://www.unix.com/man-page/Linux/2/clone/) (disponibile solo in Linux e non in MacOS, dove per creare processi leggeri “thread” si usa pthread che esiste anche sia per MacOS che Linux) e viene identificato attraverso il Process ID (PID), (in C una var di tipo pid\_t).

**Come viene creato un processo in UNIX? - Quale effetto ha la syscall fork?**

Per crearlo si usa principalmente la syscall [fork()](https://www.unix.com/man-page/Linux/2/fork/), che come per la mitosi per le cellule, quando viene eseguita il SO crea una copia identica del processo genitore, inclusi lo stato corrente del processo, la memoria, il contesto dell'utente e altre risorse, separando quindi il codice dal punto in cui avviene.

Dopo la creazione del processo figlio, entrambi i processi (genitore e figlio) iniziano ad eseguire il codice a partire dal punto in cui è stata chiamata fork. Tuttavia, ci sono differenze nella restituzione di fork tra i processi genitore e figlio, infatti ritorna 0 al processo figlio e ritorna il PID del figlio al padre.

**Come termina un processo UNIX?**

Per terminare un processo o lo si deve far terminare forzatamente tramite “kill” oppure si può usare la syscall [\_exit()](https://www.unix.com/man-page/Linux/2/exit/); questa quando chiamata prende come parametro uno **stato** che indica lo stato di uscita di un processo; solitamente si usa 0 per indicare che non ci sono stati errori, però si può personalizzare a seconda delle necessità che ha il programmatore; alcuni valori sono usati come convenzioni POSIX, e ad esempio EXIT\_SUCCESS e EXIT\_FAILURE (definiti in <stdlib.h>) sono valori standard per indicare l'esito positivo o negativo di un processo.

**A cosa serve la syscall getppid?**

La syscall [getppid()](https://www.unix.com/man-page/Linux/2/getppid/) restituisce il PID (Process ID) del processo padre; questo può essere utile quando un processo ha bisogno di identificare il suo genitore per motivi di gestione o comunicazione tra processi.

**Quale è lo scopo della syscall wait (o waitpid)?**

La [wait()](https://www.unix.com/man-page/Linux/2/wait/) (o [waitpid()](https://www.unix.com/man-page/Linux/2/waitpid/)) serve ad un processo genitore ad attendere la terminazione di un processo figlio e ha come parametro di ritorno lo stato di terminazione; in particolare, wait() attende il termine di qualsiasi processo figlio, mentre waitpid() attende un processo figlio specifico identificato dal PID. Entrambi bloccano il processo genitore fino al termine del processo figlio.

**Cosa succede alla terminazione di un processo orfano?**

Alla terminazione di un processo orfano, ovvero un processo figlio la cui terminazione avviene prima di quella del processo genitore, il processo stesso viene adottato dal processo init (PID 1). Init diventa il genitore del processo orfano e ne rilascia le risorse, evitando la formazione di processi zombie.

**Cosa sono i processi zombie e perché esistono?**

I processi zombie sono processi figli che hanno completato l'esecuzione ma il loro stato non è ancora stato raccolto dal processo genitore. Esistono perché il processo genitore non ha ancora chiamato wait() o waitpid() per leggere lo stato di uscita del processo figlio.

**Come si lancia l'esecuzione di programmi? La syscall execve e tutte le funzioni sorelle nella libreria C: exec\*:** le funzioni della famiglia exec() sono disponibili nella libreria standard del linguaggio C (<unistd.h>) e permettono di eseguire un nuovo programma. Alcune di queste includono:

* execl(): Esegue un programma specificando gli argomenti come lista di argomenti separati da virgole nella funzione stessa.
* [execve()](https://www.unix.com/man-page/Linux/2/execve/) è una delle chiamate di sistema fondamentali che sostituisce l'immagine del processo corrente con un nuovo programma. Per esempio, in C, si può utilizzare passando il percorso del programma eseguibile e gli argomenti da passare al programma e si possono specificare gli argomenti come array di stringhe (char \*argv[]).

Altre varianti come execle(), execvp(), execvpe() e altre, offrono funzionalità aggiuntive o diverse modalità di specificare il programma da eseguire e i suoi argomenti.

\*\*\* gestione file system

**Cosa è la current working directory? chdir/getcwdd**

La current working directory (directory di lavoro corrente) è la directory in cui un programma o un processo sta attualmente operando. È la directory di base da cui vengono eseguite le operazioni relative al filesystem, come la creazione o la lettura di file. Per quanto riguarda le syscall?

* chdir() (change directory) viene utilizzata per cambiare la directory di lavoro corrente in una specificata.
* getcwd() (get current working directory) restituisce il percorso della directory di lavoro corrente.

**creazione e cancellazione di directory mkdir/rmdir**

Le funzioni mkdir() e rmdir() sono comunemente utilizzate per creare e rimuovere directory vuote.

**quali sono e come funzionano le syscall per creare link fisici e simbolici?**

link() crea un nuovo collegamento diretto a un file esistente, mentre symlink() crea un collegamento simbolico a un file.

**readlink: la system call per "vedere" il target di un link simbolico**

La system call readlink() è utilizzata per leggere il percorso del target di un link simbolico in molti sistemi operativi, tra cui Linux e Unix. Questa chiamata di sistema restituisce il percorso del file o della directory a cui punta il link simbolico.

**la "cancellazione" di un file in UNIX: qual è la particolarità della syscall unlink?**

La syscall unlink() è utilizzata per rimuovere un file dal filesystem. La particolarità è che non elimina direttamente il contenuto del file, ma solo il collegamento al nome del file nel filesystem.

Quando viene chiamata unlink(), il sistema operativo rimuove il collegamento tra il nome del file e i suoi contenuti sul disco. Se il file era l'unico riferimento al suo contenuto, ovvero non era aperto da nessun processo, il filesystem lo segnala come spazio libero e il contenuto del file può essere sovrascritto in futuro da nuovi file. Se ci sono ancora processi che mantengono il file aperto, i dati del file rimarranno nel disco fino a quando tutti i processi non lo chiuderanno. Una volta che il file non ha più alcun riferimento e nessun processo lo sta utilizzando, lo spazio occupato dal file sul disco viene effettivamente liberato.

La rimozione di un file tramite `unlink()` è un'operazione irreversibile e non c'è un meccanismo diretto per recuperare un file eliminato in questo modo, a meno che non siano stati presi appositi provvedimenti di backup. È importante notare che `unlink()` non funziona per le directory. Per rimuovere una directory, è necessario utilizzare la syscall `rmdir()`, che rimuove solo directory vuote. Se si desidera rimuovere una directory e tutti i suoi contenuti, è necessario utilizzare un'operazione ricorsiva o una funzione specifica del linguaggio di programmazione che elimini tutti i file e le directory in essa contenuti prima di chiamare `rmdir()` sulla directory stessa.

**perché c'è anche la syscall rename? non è equivalente a link(vecchio, nuovo)+unlink(vecchio)?**

La syscall rename() è utilizzata per rinominare un file o spostarlo da una directory a un'altra. Anche se potrebbe sembrare che sia possibile raggiungere lo stesso risultato utilizzando link() e unlink() separatamente, ci sono diverse ragioni per cui è preferibile utilizzare rename():

se si desidera rinominare o spostare un file, è consigliabile utilizzare la syscall rename() per garantire:

* atomicità: potrebbe verificarsi una situazione in cui un processo vede temporaneamente il file vecchio ma non il nuovo o viceversa
* efficienza: non coinvolge la duplicazione del contenuto del file o la modifica dei puntatori dei blocchi di dati sul disco, ma modifica solo i metadati del filesystem per rinominare il file o spostarlo

Dunque, link() e unlink() possono essere utili per creare link fisici tra file o per eliminare file specifici, non sono equivalenti a rename() quando si tratta di operazioni di rinomina o spostamento di file.

**La system call per leggere le informazioni dei file: stat**

la system call stat() è utilizzata per ottenere informazioni dettagliate sui file. Questa chiamata di sistema prende come argomento il percorso del file e restituisce una struttura di dati che contiene varie informazioni sul file, come ad esempio Tipo di file (regolare, directory, link simbolico, etc.), dimensione del file in byte, informazioni sui permessi del file (proprietario, gruppo, altri) e Data e ora di ultima modifica, accesso e cambiamento dei metadati del file.

**Creare file speciali: mknod**

mknod() è utilizzata per creare file speciali in sistemi Unix-like. Questi file speciali possono essere di diversi tipi, tra cui:

* Dispositivi di blocco (S\_IFBLK): rappresentano dispositivi di storage che trasferiscono dati in blocchi fissi, come ad esempio un disco rigido.
* Dispositivi di carattere (S\_IFCHR): rappresentano dispositivi che trasferiscono dati carattere per carattere, come ad esempio una tastiera o un mouse.
* Sockets (S\_IFSOCK): rappresentano endpoint per la comunicazione tramite socket.
* Pipe (S\_IFIFO o "named pipe"): rappresentano canali di comunicazione tra processi.

**cambiare i permessi di un file: chmod**

chmod() è utilizzata per cambiare i permessi di accesso di un file in sistemi Unix-like. Questa chiamata di sistema consente di modificare i permessi di lettura, scrittura ed esecuzione del file per il proprietario, il gruppo e gli altri utenti

**cambiare il proprietario/gruppo di un file: chown**

chown() è utilizzata per cambiare il proprietario e/o il gruppo di un file in sistemi Unix-like. Questa chiamata di sistema consente di modificare il proprietario e/o il gruppo di un file a livello di SO

**access: controllo a priori se una operazione su un file è permessa: perché questa system call è deprecata?**

La syscall `access()` è stata considerata deprecata in alcuni contesti a causa di possibili problemi di sicurezza e di mancanza di affidabilità nel determinare effettivamente se un'operazione su un file sarà permessa o meno. Ecco alcuni motivi principali per cui `access()` è considerata deprecata:

* Race condition: verifica i permessi di accesso a un file in base allo stato del file system al momento della chiamata. Tuttavia, durante il breve intervallo di tempo tra la chiamata a `access()` e l'effettiva operazione sul file (ad esempio, l'apertura o la lettura del file), lo stato del file system potrebbe cambiare; dunque in un Attacco TOCTOU "Time-of-check to time-of-use”; un attaccante potrebbe sostituire un file dopo che `access()` ha verificato i permessi, ma prima che l'operazione effettiva venga eseguita, portando a un risultato imprevisto o addirittura a una violazione di sicurezza;
* Privilegi non garantiti: Anche se `access()` indica che un'operazione è permessa, non ciò garantisce che l'operazione avrà successo. Ciò può accadere se ad esempio il file viene eliminato poco dopo la verifica dei permessi.

A causa di questi problemi, molte implementazioni e ambienti di sviluppo preferiscono metodi alternativi per verificare i permessi di accesso a un file. Ad esempio, in ambiente Unix-like, è comune utilizzare la syscall `open()` con gli appositi flag per verificare i permessi di accesso e, contemporaneamente, aprire il file, evitando così le race condition associate a `access()`.

**Quale è lo scopo della system call umask?**

umask() consente a un processo di impostare i permessi predefiniti dei file e delle directory che crea. Immagina la umask come una sorta di filtro che rimuove certi permessi quando un nuovo file o una nuova directory vengono creati.

Quando un processo crea un nuovo file o una nuova directory, il SO applica automaticamente la umask per rimuovere alcuni di questi permessi predefiniti. Ad esempio, se la umask è impostata a 022, significa che il permesso di scrittura sarà rimosso per gruppo e altri, quando vengono creati nuovi file o directory.

**aggiungere/togliere sottoalberi al file system: mount/umount2**

* mount(): estende l'albero del filesystem attuale aggiungendo il contenuto di un filesystem esterno (come un dispositivo esterno o una partizione) in una directory specifica del filesystem locale. Ad esempio, puoi montare un dispositivo USB in una directory del tuo sistema operativo, rendendo così i file contenuti nel dispositivo accessibili tramite il filesystem del tuo computer;
* umount2(): smonta un filesystem montato precedentemente dal filesystem locale. Quando smonti un filesystem, rimuovi l'albero del filesystem esterno dalla directory specificata, rendendo i suoi file inaccessibili tramite il filesystem del tuo computer.

È importante notare che queste syscall richiedono privilegi di amministratore per essere eseguite, in quanto operano a livello di sistema e possono influenzare l'intero filesystem.

Infine, `mount -a` è un comando utilizzato per montare tutti i filesystems elencati in `/etc/fstab`, che è un file di configurazione che elenca i filesystems montati automaticamente all'avvio del sistema. Questo comando è utile per montare tutti i filesystems specificati in `/etc/fstab` dopo un riavvio del sistema.

**cambiare l'ampiezza di un file: truncate. Cosa succede se su un file si eseguono in successione open(..., O\_CREAT), truncate, close?**

La syscall `truncate()` viene utilizzata per ridimensionare un file esistente o per creare un nuovo file di dimensione specifica. Se si specifica un file esistente e si riduce la dimensione, i dati oltre la nuova dimensione vengono eliminati. Se si specifica un nuovo file e una dimensione maggiore di zero, viene creato un nuovo file con la dimensione specificata. Tuttavia, `truncate()` non viene utilizzata per creare dischi.

Se esegui in successione le syscall open(), truncate(), e close() su un file, ecco cosa succede:

1. open(..., O\_CREAT): Questa chiamata di sistema apre il file specificato. Se il file non esiste, viene creato. Se esiste già, la chiamata open() semplicemente apre il file. La flag O\_CREAT indica che se il file non esiste, deve essere creato.
2. truncate: Questa chiamata di sistema ridimensiona il file aperto dalla chiamata open() precedente. Se il file non esiste, verrà creato con la dimensione specificata. Se il file esiste già, la sua dimensione verrà ridotta o aumentata al valore specificato.
3. close(): Questa chiamata di sistema chiude il file precedentemente aperto con open(). Quando il file viene chiuso, tutte le risorse associate ad esso vengono liberate e il file non è più accessibile per la scrittura o la lettura fino a quando non viene riaperto.

**La system call scomoda e per ottenere la lista dei file di una directory: getdents.**

La syscall getdents() è utilizzata per ottenere la lista dei file all'interno di una directory. È considerata "scomoda" perché richiede una gestione più basso livello rispetto ad altre interfacce di più alto livello come ‘ls’.

**Cambiare i tempi dei file: utime**

utime() è utilizzata per cambiare i tempi di accesso e modifica di un file. Essa permette di modificare i timestamp di accesso e modifica di un file specificato nel percorso pathname.

**Le varianti 'l-' delle syscall: lstat, lchown, lchmod**

Le varianti 'l-' delle syscall operano su collegamenti simbolici anziché sui file a cui i collegamenti simbolici puntano. Ecco una breve spiegazione di ciascuna:

1. lstat(): simile alla syscall `stat()`, ma invece di restituire le informazioni sul file a cui il percorso specificato punta, restituisce le informazioni sul collegamento simbolico stesso. Ad esempio, restituisce le informazioni sul collegamento simbolico se si fornisce il percorso di un collegamento simbolico come argomento.
2. lchown():simile alla syscall `chown()`, ma cambia il proprietario e/o il gruppo del file a cui il percorso specificato punta, cambia il proprietario e/o il gruppo del collegamento simbolico stesso.
3. lchmod(): simile alla syscall `chmod()`, ma invece di cambiare i permessi del file a cui il percorso specificato punta, cambia i permessi del collegamento simbolico stesso.

\*\*\* gestione file

**La syscall open, come funziona e perché può avere 2 o 3 parametri?**

La [open()](https://www.unix.com/man-page/Linux/2/open/) è utilizzata per aprire o creare un file e restituisce un file descriptor che può essere usato per leggere, scrivere o manipolare il file. Se chiamata con due parametri permette di prendere in input:

* pathname è il percorso del file da aprire o creare.
* flags sono i flag che determinano il modo in cui il file deve essere aperto o creato, come ad esempio `O\_RDONLY` per aprire il file in sola lettura, `O\_WRONLY` per aprire il file in sola scrittura, o `O\_RDWR` per aprire il file in lettura/scrittura.

Tuttavia, `open()` può anche avere un terzo parametro opzionale:

* `mode` sono i permessi che devono essere applicati al file se viene creato. Questo parametro viene ignorato se il file esiste già. Per i dettagli sui permessi, si può fare riferimento alla syscall `chmod()`.

duplicazione di file system: dup/dup2/dup3

**Cosa succede ai file aperti quando un processo esegue una fork?**

Quando un processo esegue una fork, tutti i file aperti dal processo padre vengono ereditati dal processo figlio. Ciò significa che il processo figlio avrà accesso agli stessi file descriptor e ai file o alle risorse a cui essi puntano.

**Cosa succede ai file aperti quando un processo esegue una exec?**

Quando un processo esegue una exec, il processo corrente viene sostituito con un nuovo programma. Durante questo processo di sostituzione, lo stato del processo, inclusi i file aperti, viene completamente riassegnato al nuovo programma. Ciò significa che tutti i file aperti dal processo precedente vengono chiusi automaticamente durante l'esecuzione di exec, a meno che non siano stati aperti con il flag FD\_CLOEXEC impostato, che indica al kernel di chiudere automaticamente il file descriptor quando viene eseguito un exec.

**Scopo della syscall close.**

La syscall close() è utilizzata per chiudere un file descriptor aperto da un processo. Quando un file descriptor viene chiuso con successo, il SO libera le risorse associate al file descriptor e lo rende disponibile per un eventuale riutilizzo.

**Qual è l'uso delle syscall read e write?**

Le syscall read e write sono fondamentali per la gestione dell'I/O nei sistemi Unix.

* read: legge dati da un file descriptor nell'area di memoria specificata.
* write: scrive dati dall'area di memoria specificata su un file descriptor.

Queste syscall consentono ai processi di leggere e scrivere dati da e verso i file o i dispositivi di I/O, consentendo l'interazione con il filesystem, le periferiche di rete e altri dispositivi di input/output

**Come vengono gestiti i buffer multipli di readv e writev.**

Le syscall `readv` e `writev` in Unix consentono operazioni di lettura e scrittura su più buffer contemporaneamente, ottimizzando l'I/O. Quando vengono utilizzate, i buffer specificati vengono letti o scritti in sequenza.

Per la lettura (`readv`), il kernel legge dati da più buffer in successione fino a quando non ha letto la quantità richiesta di dati o fino a quando non esaurisce i buffer.

Per la scrittura (`writev`), il kernel scrive dati su più buffer in successione fino a quando non ha scritto la quantità richiesta di dati o fino a quando non esaurisce i buffer.

Queste syscall sono utili per ottimizzare l'operazione di lettura/scrittura di dati da e verso più buffer contemporaneamente, riducendo il numero di chiamate di sistema necessarie e migliorando le prestazioni complessive del sistema.

**Come funzionano pread e pwrite e che senso hanno nei programmi multithread**

Le syscall `pread` e `pwrite` in Unix consentono la lettura e la scrittura di dati da o verso un file a una posizione specificata senza modificare l'offset del file. Questo è utile quando si desidera leggere o scrivere dati in modo atomico senza preoccuparsi delle modifiche concorrenti all'offset del file da parte di altri thread.

Nei programmi multithread, l'uso di `pread` e `pwrite` può essere preferibile rispetto a `read` e `write` quando più thread accedono simultaneamente allo stesso file. Utilizzando `pread` e `pwrite`, ciascun thread può specificare l'offset dal quale leggere o scrivere i dati senza preoccuparsi delle modifiche all'offset da parte degli altri thread. Ciò aiuta a garantire coerenza e atomicità nelle operazioni di I/O, riducendo il rischio di corruzione dei dati o condizioni di gara tra i thread.

**Qual è il funzionamento di lseek? Come vengono gestiti le lseek oltre la fine del file?**

La syscall lseek in Unix consente di spostare l'offset di lettura/scrittura di un file. Il parametro offset specifica la nuova posizione dell'offset, mentre whence specifica come interpretare l'offset rispetto ad altri punti nel file (inizio, corrente o fine del file).

Se lseek viene utilizzata per spostare l'offset oltre la fine del file durante una scrittura, il file viene esteso per includere l'offset specificato. Se si esegue una lettura oltre la fine del file, viene restituito un valore di fine file (EOF) quando il programma tenta di leggere dati oltre l'attuale fine del file.

**Quali flag possono essere gestiti con fcntl?**

La syscall `fcntl` in Unix viene utilizzata per eseguire varie operazioni su un descrittore di file, tra cui la gestione dei flag. Alcuni dei flag che possono essere gestiti con `fcntl` includono:

1. F\_GETFL`: Restituisce i flag del file associati al descrittore.

2. `F\_SETFL`: Imposta i flag del file associati al descrittore.

3. `O\_NONBLOCK`: Imposta il file descriptor in modalità non bloccante.

4. `O\_APPEND`: Imposta l'apertura del file in modalità di scrittura in append.

5. `O\_ASYNC`: Abilita le notifiche di I/O asincrone per il file descriptor.

6. `O\_DIRECT`: Specifica l'I/O diretto, bypassando la cache del kernel.

7. `O\_SYNC`: Richiede che le operazioni di scrittura siano sincrone.

**Funzioni specifiche dei file speciali (device): uso della syscall ioctl.**

La syscall `ioctl` in Unix viene utilizzata per eseguire operazioni specifiche su file speciali, come i dispositivi. Alcuni usi comuni di `ioctl` includono:

1. Controllo del comportamento del dispositivo: Per configurare parametri o modalità di funzionamento del dispositivo.

2. Controllo dei flag del dispositivo: Per abilitare o disabilitare opzioni specifiche del dispositivo.

3. Comunicazione con il driver del dispositivo: Per inviare comandi o richieste specifiche al driver del dispositivo.

4. Recupero di informazioni dal dispositivo: Per ottenere dati o informazioni sullo stato del dispositivo.

5. Gestione della cache: Per eseguire operazioni di gestione della cache del dispositivo, come la pulizia o l'invalidazione della cache.

Questi sono solo alcuni esempi di come `ioctl` possa essere utilizzata per interagire con file speciali, come dispositivi hardware, in un sistema Unix. La specifica funzionalità e il comportamento di `ioctl` possono variare a seconda del tipo di dispositivo e del driver associato.

**Le varianti 'f-' delle syscall di gestione del file system: fstat, fchown, fchmod, ftruncate, fchdir**

Gestiscono file specifici identificati da un descrittore di file anziché da un percorso nel file system:

1. fstat: Restituisce informazioni sul file associate al descrittore di file.

2. fchown: Cambia il proprietario e il gruppo del file identificato dal descrittore di file.

3. fchmod: Cambia i permessi del file identificato dal descrittore di file.

4. ftruncate: Riduce o estende la dimensione del file identificato dal descrittore di file.

5. fchdir: Cambia la current working directory del processo chiamante nel percorso del file identificato dal descrittore di file.

Queste syscall operano sui file aperti nel processo corrente, consentendo modifiche dirette senza la necessità di specificare nuovamente il percorso del file.

**l'opzione O\_PATH di open:**

viene utilizzata per aprire un file senza ottenere accesso in lettura o scrittura, ma solo per ottenere un descrittore di file valido per il file stesso; può essere utile quando si desidera ottenere informazioni sul file o eseguire operazioni su di esso, come la lettura del link simbolico associato o il recupero delle informazioni sul file, senza dover aprire il file per la lettura o la scrittura. Ciò consente di ridurre al minimo l'accesso al file stesso durante l'esecuzione di operazioni di sistema.

**le varianti con suffisso -at: openat, mkdirat, mknodat, fchownat, fchmodat, fstatat, unlinkat, linkat, symlinkat, renameat, faccessat...**

Le varianti con il suffisso -at delle syscall, come openat, mkdirat, renameat, ecc., consentono di specificare percorsi relativi rispetto a un determinato directory file descriptor (fd) anziché utilizzare percorsi assoluti. Questo offre maggiore flessibilità e controllo nei programmi multithread o nei programmi che devono operare su più directory contemporaneamente.

\*\*\* gestione utenti

**UNIX prevede 3 tipi di utente e di gruppo: reale, effettivo e salvato: perché?**

servono a gestire in modo flessibile le autorizzazioni e le identità dei processi:

* Utente reale (real user): Si riferisce all'identità dell'utente che ha effettuato l'accesso al sistema. È l'utente che ha effettuato il login tramite il nome utente e la password.
* Utente effettivo (effective user): Indica l'identità dell'utente che il processo attualmente esegue. Questo può essere differente dall'utente reale a causa di meccanismi come i setuid bit o setgid bit sui file eseguibili. Quando un file ha questi bit abilitati, il processo che lo esegue assume l'identità dell'utente proprietario del file anziché dell'utente che ha effettuato l'accesso.
* Utente salvato (saved user): Questo concetto si riferisce all'utente precedente quando si effettua un cambio temporaneo dell'utente effettivo. Viene memorizzato per garantire che un processo possa "tornare indietro" al suo stato utente precedente dopo aver temporaneamente cambiato il suo utente effettivo.

L'utente "root" è un utente speciale nei sistemi UNIX che ha tutti i privilegi e l'accesso completo al sistema. È in grado di eseguire qualsiasi operazione, modificare qualsiasi file e cambiare l'identità di qualsiasi processo, il che lo rende potenzialmente pericoloso se non utilizzato correttamente.

**getuid, setuid, getgid, setgid, geteuid, seteuid, getreiud, setreuid, getresuid, setresuid, getresgid, setresgid, getgroups, setgroups**

Queste sono tutte syscall utilizzate per gestire l'identità e i privilegi degli utenti e dei gruppi in un processo su un sistema UNIX-like. Ad esempio, getuid restituisce l'ID utente effettivo del processo, setuid imposta l'ID utente effettivo, getgid restituisce l'ID del gruppo effettivo, e così via. Le varianti con e indicano gli ID effettivi, mentre res si riferisce agli ID residui (utilizzati per le operazioni di cambio identità). getgroups restituisce l'elenco dei gruppi supplementari di un processo, mentre setgroups li imposta.

\*\*\* segnali

**Le syscall kill e signal hanno nomi fuorvianti: cosa fanno?**

* Syscall [kill](https://www.unix.com/man-page/Linux/2/kill/): Questa syscall consente a un processo di inviare un segnale a un altro processo o a se stesso nel sistema operativo. Nonostante il nome possa far pensare che "killi" un processo, in realtà può essere utilizzata per inviare diversi tipi di segnali a un processo specifico. Per esempio, può essere usata per terminare un processo inviando il segnale SIGKILL, ma può anche essere utilizzata per notificare il processo di eventi specifici come la richiesta di terminazione (SIGTERM).
* Syscall [signal](https://www.unix.com/man-page/Linux/2/signal/): Questa syscall è utilizzata per configurare la gestione dei segnali per un processo. Anche se il suo nome può far pensare a un'azione diretta di invio di segnali, in realtà è utilizzata per assegnare un gestore (una funzione o un comportamento) a un determinato segnale all'interno di un processo. In altre parole, consente a un processo di stabilire come vuole gestire un determinato segnale quando lo riceve.

Segmentation fault è un esempio di segnale, che capita quando si vuole accedere in memoria ad un indirizzo sbagliato (magari puntando a qualcosa usando NULL); quando il kernel vuole comunicare col processo usa i segnali.

Sigaction è più flessibile di signal. Cosa permette in più?

Le funzioni di gestione dei segnali.

Cosa sono le funzioni signal-safe? Perché esistono funzioni unsafe?

La definizione di insiemi di segnali sigsetops

Mascheramento e attesa di segnali (sigsuspend, sigprocmask, sigpending)

\*\*\* comunicazione fra processi (locali)

pipe

named pipe

(unix sockets)

\*\*\* attesa di eventi

ieri: select, pselect

oggi: poll, ppoll

domani epoll (solo linux).

\*\*\* Networking: Berkeley sockets

socket, bind, listen, accept, connect, sendto, recvfrom, sendmsg, recvmsg, sengmmsg, recvmmsg, shutdown, getsockname, getpeername, socketpair, getsockopt, setsockopt

\*\*\* altre syscall:

gestione del tempo: time, gettimeofday, settimeofday, nanosleep, setitimer, getitimer

memory mapping: mmap, mremap, munmap

usi speciali di file descriptor: eventfd, signalfd, pidfd, timerfd...

gurdie sul file system: inotify

Il file system è una astrazione di un archivio e tutte le modalità di accesso al file prevedono l’apertura, ci ritorna sempre un descrittore che serve per effettuare la stessa op; le sys call principali sono:

…

* 231123

La fchownat è utilizzata per cambiare l'identità del proprietario e il gruppo di un file specificato da un percorso relativo a un file descriptor. Questo significa che fchownat richiede un file descriptor (ottenuto tramite open o funzioni simili) e un percorso relativo al file all'interno del file system associato a quel file descriptor. È utile quando si desidera cambiare i permessi di un file aperto senza dover fare riferimento al percorso del file stesso.

chown: Utilizzato per cambiare il proprietario di un file o di una directory.

La principale differenza è quindi che chown è un comando di shell utilizzato per cambiare il proprietario dei file o delle directory, mentre fchownat è una chiamata di sistema utilizzata da programmi scritti in linguaggi come C per cambiare i permessi di un file utilizzando un file descriptor.

In sostanza, mentre entrambi hanno lo stesso obiettivo di cambiare i permessi dei file, chown è utilizzato direttamente dalla shell, mentre fchownat è utilizzato all'interno del codice dei programmi.

chmod: Usato per modificare i permessi di lettura, scrittura ed esecuzione di un file o di una directory.

\*\*\* segnali

Le syscall kill e signal hanno nomi fuorvianti:

* Syscall [kill](https://www.unix.com/man-page/Linux/2/kill/): Questa syscall consente a un processo di inviare un segnale a un altro processo o a se stesso nel sistema operativo. Nonostante il nome possa far pensare che "killi" un processo, in realtà può essere utilizzata per inviare diversi tipi di segnali a un processo specifico. Per esempio, può essere usata per terminare un processo inviando il segnale SIGKILL, ma può anche essere utilizzata per notificare il processo di eventi specifici come la richiesta di terminazione (SIGTERM).
* Syscall [signal](https://www.unix.com/man-page/Linux/2/signal/): Questa syscall è utilizzata per configurare la gestione dei segnali per un processo. Anche se il suo nome può far pensare a un'azione diretta di invio di segnali, in realtà è utilizzata per assegnare un gestore (una funzione o un comportamento) a un determinato segnale all'interno di un processo. In altre parole, consente a un processo di stabilire come vuole gestire un determinato segnale quando lo riceve.

Segmentation fault è un esempio di segnale, che capita quando si vuole accedere in memoria ad un indirizzo sbagliato (magari puntando a qualcosa usando NULL); quando il kernel vuole comunicare col processo usa i segnali

Il segnale SIGSEGV è una notifica che indica che un programma ha tentato di accedere a una parte della memoria a cui non ha il permesso di accedere. Questo può accadere per diverse ragioni, come ad esempio:

* Dereferenziare un puntatore NULL: Quando un programma cerca di accedere o manipolare dati attraverso un puntatore che non punta a nessuna area di memoria valida.
* Accesso a memoria non allocata o non autorizzata: Un programma potrebbe tentare di accedere a una parte della memoria che non è stata allocata per esso o che è protetta dalle autorizzazioni di accesso.
* Buffer overflow o sottoscrizione degli array: Accedere a un'area di memoria oltre i limiti consentiti per un array può causare questo segnale.

SIGCHILD avviene quando un figlio ha un cambio di stato, ad esempio la sua terminazione

Sigaction è più flessibile di signal. Cosa permette in più?

Le funzioni di gestione dei segnali.

Cosa sono le funzioni signal-safe? Perché esistono funzioni unsafe?

La definizione di insiemi di segnali sigsetops

Mascheramento e attesa di segnali (sigsuspend, sigprocmask, sigpending)

\*\*gestione tempo ///1 gennaio 1970 avanti UNIX e dopo UNIX

time ritorna un intero a 64 bit di tipo time\_t con il numero dei secondi dal 1 gennaio 1970

I file descriptor sono identificatori numerici utilizzati per accedere ai file aperti da un processo in un sistema Unix o Unix-like. Questi numeri vengono assegnati ai file aperti, alle pipe, ai socket, ai dispositivi di input/output (ad esempio, terminali, stampanti) e ad altri stream di dati.

Ecco alcuni concetti chiave relativi ai file descriptor:

1. **Numeri interi:** I file descriptor sono numeri interi non negativi. Di solito, i primi tre file descriptor (0, 1 e 2) sono predefiniti per rappresentare rispettivamente lo standard input (**stdin**), lo standard output (**stdout**) e lo standard error (**stderr**) di un processo.
2. **Apertura e chiusura dei file:** Quando un processo apre un file o un altro stream di dati, il kernel assegna un file descriptor al file aperto e restituisce il numero identificativo al processo.
3. **Gestione dei file descriptor:** Un processo può gestire più file descriptor contemporaneamente. Può aprirli, chiuderli, leggere da essi o scrivere in essi utilizzando chiamate di sistema specifiche come **open()**, **close()**, **read()**, **write()** e altre. La tabella dei file descriptor è mantenuta dal kernel e contiene le associazioni tra i file aperti e i loro file descriptor.
4. **Ereditarietà:** Quando un processo genera un nuovo processo (ad esempio, attraverso **fork()**), il nuovo processo eredita i file descriptor del processo genitore. Tuttavia, possono esistere meccanismi per modificare o riorientare i file descriptor nel processo figlio.
5. **Limiti e risorse:** I sistemi operativi hanno limiti sul numero massimo di file descriptor che un processo può aprire contemporaneamente. Questi limiti possono essere modificati e controllati dal sistema operativo o dalle impostazioni del sistema.

I file descriptor sono fondamentali nel sistema Unix per consentire ai processi di interagire con i file e altre risorse di I/O. Consentono ai programmi di accedere a file e dispositivi di input/output in modo efficiente e consistente, fornendo un'astrazione standard per l'I/O nei sistemi Unix.

Segnali:

Vanno nel process control block di un processo e vengono posti li in attesa di essere esaminati; significa dirgli di fare qualcosa; sono documentati nel “man 7”

Un segnale ha una default action, ossia:

- IGN, ossia ignora il segnale come se non lo avesse ricevuto;

- TERM fa terminare il processo

- TERM con CORE DUMP --> termina e fa un dumping della memoria

- STOP manda il processo nello stato di stopped

- CONT sblocca un processo bloccato con STOP

Si può definire cosa fare quando si riceve un segnale sovrascrivendo le default actions (alcuni tipo sigkill no)

Ci sono vari segnali nel man 7 alla quale sono associati una di queste defaul actions;

Un programma può lancuiare un segnale solo se si hanno i processi necessari (per ragioni di sicurezza)